REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
	Final Report	17 September 1996 - 11-Jun-97

4. TITLE AND SUBTITLE

Programming Environment mpC for Distributed Memory Machines

5a. CONTRACT NUMBER
F6170896W0319

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Professor Victor Ivannikov

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Russian Academy of Sciences
Bolshaya Communisticheskaya 25
Moscow 109004
Russia

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0014

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
SPC 96-4061

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report results from a contract tasking Russian Academy of Sciences as follows: The contractor will investigate portable parallel programming language. He will investigate new parallel programming language concepts for general purpose parallel computing; develop a prototype parallel programming environment that addresses large and fine-grain parallelism, control-parallel and data-parallel paradigms, and static and dynamic task creation; and devise workable mechanisms to permit collaboration by US and Russian scientists over the Internet.

15. SUBJECT TERMS
EOARD, Computers

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18, NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Jerry J. Sellers, Maj, USAF
UNCLAS	UNCLAS	UNCLAS	UL	21	19b. TELEPHONE NUMBER *(Include area code)* +44 (0)20 7514 4318

Programming Environment mpC for Distributed Memory Machines (Report on Special Contract SPC-96-4061)

Authors: Alexey L. Lastovetsky (project leader)
Alexey Ya. Kalinov (senior researcher)
Ilya N. Ledovskih (researcher)
Dmitry M. Arapov (researcher)
Mikhail A. Posypkin (junior researcher)

Russian Academy of Sciences
Institute for System Programming
25 Bolshaya Kommunisticheskaya str.
Moscow 109004
Russia

Director: Prof. Victor P. Ivannikov
Phone: 7(095)912-4425
Fax: 7(095)912-1524
E-mail: ivan@ispras.ru

Moscow
January 15th, 1997

1. Introduction

The mpC language and its programming environment was initially developed to support programming for massively parallel computers, first of all for high-performance distributed memory machines (DMMs). In brief, our motivation of mpC was as follows.

Programming for DMMs is based mostly on message-passing function extensions of C or Fortran, such as PVM and MPI. But it is tedious and error-prone to program in a message-passing language, because of its low level. Therefore, high-level languages that facilitate parallel programming have been developed for DMMs. They can be divided into two classes depending on the parallel programming paradigm - task parallelism or data parallelism - underlying them. Task parallel and data parallel programming languages allow the user to implement different classes of parallel algorithms. But efficient implementation of many problems needs parallel algorithms that can not be implemented in pure data parallel or task parallel styles. We have developed the mpC language (as an ANSI C superset) which supports both task and data parallelism, allows both static and dynamic process and communication structures, enables optimizations aimed at both communication and computation, and supports modular parallel programming and the development of a library of parallel programs.

The mpC language is based on the notion of network consisting of virtual processors of different types and performances connected with links of different bandwidths. The user can describe network topology, create and discard networks, and distribute data and computations over the networks. That is, the user can specify (dynamically!) in details virtual parallel machine which performs his application.

In other words, the user can specify the topology of his application, and the programming environment will use this (topological) information **in run time** to ensure the efficient execution of the application on any particular DMM.

Currently, a prototype programming environment includes a compiler, a run-time support system, a library, and a command-line user interface.

The compiler translates a source mpC program into ANSI C code with calls to functions of the run-time support system.

Run-time support system manages the computing space which consists of a number of processes running over target DMM as well as provides communications. It has a precisely specified interface and encapsulates a particular communication package (currently, a small subset of MPI). It ensures platform-independence of the rest of system components.

The library consists of a number of functions which support debugging mpC programs as well as provide some low-level efficient facilities.

The command-line user interface consists of a number of shell commands supporting the creation of a virtual DMM and the execution of mpC programs on the machine. While creating the machine, its topology is detected by a topology detector running a special benchmark and saved in a file used by the run-time support system.

When developing the mpC programming environment, we used a network of workstations running MPI as a target parallel machine and found, that the principles, on which mpC is based, make this programming language and its programming environment be very convenient tools for development of efficient and portable parallel programs for heterogenous networks of workstations.

The point is that all programming environments for DMMs which we know of have one common property. Namely, when developing a parallel program, either the user has no facilities to describe the virtual parallel system executing the program, or such facilities are too poor to spec-

ify an efficient distribution of computations and communications over the target DMM. Even topological facilities of MPI (as well as MPI-2) have turned out insufficient to solve the problem. So, to ensure the efficient execution of the program on a particular DMM, the user must use facilities which are external to the program, such as boot schemes and application schemes. If the user is familiar with both the topology of target DMM and the topology of the application, then, by using such configurational files, he can map the processes which constitute the program onto processors which make up DMM, to provide the most efficient execution of the program. But if the application topology is defined in run time (that is, if it depends on input data), it won't be successful.

The mpC language allows the user to specify an application topology, and its programming environment uses the information in run time to map processes onto processors of target DMM resulting in efficient execution of the application.

Section 2 of the paper outlines the mpC language. Section 3 sketches the prototype programming environment. Section 4 demonstrates how mpC may be used to develop efficient and portable irregular applications for DMMs. Section 5 demonstrates how mpC may be used to develop efficient and portable regular applications for heterogeneous DMMs. In addition, sections 4 and 5 tell more about the language.

2. Outline of the mpC language

In mpC, the notion of *computing space* is defined as a set of typed virtual processors of different performance connected with links of different bandwidth accessible to the user for management. There are several processor types, but most common virtual processors are of the `scalar` type. A virtual processor has an attribute characterizing its relative performance. A directed *link* connecting two virtual processors is a one-way channel for transferring data from source processor to the processor of destination.

The basic notion of the mpC language is *network object* or simply *network*. Network comprises virtual processors of different types and performances connected with links of different bandwidths. Network is a region of the computing space which can be used to compute expressions and execute statements.

Allocating network objects in the computing space and discarding them is performed in similar fashion to allocating data objects in the storage and discarding them. Conceptually, creation of new network is initiated by a virtual processor of some network already created. This virtual processor is called a *parent* of the created network. The parent belongs to the created network. The only virtual processor defined from the beginning of program execution till program termination is the pre-defined virtual *host-processor* of the `scalar` type.

Every network declared in an mpC program has a type. The type specifies the number and types and performances of virtual processors, links between these processors and their lengths characterizing bandwidths, as well as separates the parent. For example, the type declaration

```
/*1*/      nettype Rectangle {
/*2*/        coord I=4;
/*3*/        node {
/*4*/          I<2 : fast scalar;
/*5*/          I>=2: slow scalar;
/*6*/        };
/*7*/        link {
/*8*/          I>0:  [I]<->[I-1];
/*9*/          I==0: [I]<->[3];
```

```
/*10*/        };
/*11*/        parent [0];
/*12*/      };
```

introduces network type `Rectangle` that corresponds to networks consisting of 4 virtual processors of the `scalar` type and different performances interconnected with undirected links of the normal length in a rectangular structure.

In this example, line 1 is a *header* of the network-type declaration. It introduces the name of the network type.

Line 2 is a *coordinate declaration* declaring the coordinate system to which virtual processors are related. It introduces integer coordinate variable `I` ranging from 0 to 3.

Lines 3-6 are a *node declaration*. It relates virtual processors to the coordinate system declared and declares their types and performances. Line 4 stands for the predicate ***for all*** *`I<4`* ***if*** *`I<2`* ***then*** *fast virtual processor of the* `scalar` *type is related to the point with coordinate* `[I]`. Line 5 stands for the predicate ***for all*** *`I<4`* ***if*** *`I>=2`* ***then*** *slow virtual processor of the* `scalar` *type is related to the point with coordinate* `[I]`. Performance specifiers `fast` and `slow` specify relative performances of virtual processors of the same type. For any network of this type, this information allows the compiler to associate a weight with each virtual processor of the network normalizing it in respect to the weight of the parent. Note, that the virtual host-processor is always of the `scalar` type and normal performance.

Lines 7-10 are a *link declaration*. It specifies links between virtual processors. Line 8 stands for the predicate ***for all*** *`I<4`* ***if*** *`I>0`* ***then*** *there exists undirected link of normal length connecting virtual processors with coordinates* `[I]` *and* `[I-1]`, and line 9 stands for the predicate ***for all*** *`I<4`* ***if*** *`I==0`* ***then*** *there exists undirected link of normal length connecting virtual processors with coordinates* `[I]` *and* `[3]`. Note, that if a link between two virtual processors is not specified explicitly, it is meant not absence of a link but existence of a very long link.

Line 11 is a *parent declaration*. It specifies that the parent has coordinate [0].

With the network type declaration, the user can declare a network identifier of this type. For example, the declaration

```
net Rectangle r1;
```

introduces identifier `r1` of network.

The notion of *distributed data object* is introduced in the spirit of C* and Dataparallel C. Namely, a data object distributed over a region of the computing space comprises a set of components of any one type so that each virtual processor of the region holds one component. For example, the declarations

```
net Rectangle r2;
int [*]Derror, [r2]Da[10];
float [host]f, [r2:I<2]Df;
repl [*]Di;
```

declare:

- integer variable `Derror` distributed over the entire computing space;
- integer 10-member array `Da` distributed over the network `r2`;
- undistributed ⁿoating variable `f` belonging to the virtual host-processor;
- ⁿoating variable `Df` distributed over a subnetwork of network `r2`;
- integer variable `Di` replicated over the entire computing space.

By definition, a distributed object is *replicated* if all its components is equal to each other.

The notion of *distributed value* is introduced similarly.

In addition to a network type, the user can declare a parametrized family of network types called

topology or *generic network type*. For example, the declaration

```
/*1*/     nettype Ring(n, p[n]) {
/*2*/       coord I=n;
/*3*/       node {
/*4*/         I>=0: fast*p[I] scalar;
/*5*/       };
/*6*/       link {
/*7*/         I>0:  [I]<->[I-1];
/*8*/         I==0: [I]<->[n-1];
/*9*/       };
/*10*/      parent [0];
/*11*/    };
```

introduces topology `Ring` that corresponds to networks consisting of `n` virtual processors of the `scalar` type interconnected with undirected links of normal length in a ring structure.

The header (line 1) introduces parameters of topology `Ring`, namely, integer parameter `n` and vector parameter `p` consisting of `n` integers.

Correspondingly, coordinate variable `I` ranges from `0` to `n-1`, line 4 stands for the predicate ***for all*** *`I<n`* ***if*** *`I>=0`* ***then*** *fast virtual processor of the `scalar` type, whose relative performance is specified by the value of `p[I]`, is related to the point with coordinate `[I]`*, and so on.

Here, performance specifier `fast*p[I]` includes so-called power specifier `*p[I]`. In general, the value of the expression in a power specifier shall be positive integer. Any operand in the expression should consist only of coordinate variables, constants and generic parameters. If the value of the expression is equal to 1, the power specifier may be omitted.

It is meant that in the framework of the same network-type declaration any performance specifier with the `fast` keyword specifies more powerful virtual processor than a performance specifier with the `slow` keyword. It is meant also that the greater value of the expression in a power specifier the more performance is specified.

With the topology declaration, the user can declare a network identifier of a proper type. For example, the fragment

```
repl m, n[100];
/* Computing m, n[0],...,n[m-1] */
{
  net Ring(m,n) rr;
  ...
}
```

introduces identifier `rr` of the network, the type of which is defined completely only in run time. Network `rr` consists of `m` virtual processors the relative performance of `i`-th virtual processor being characterized by the value of `n[i]`.

A network has a computing space duration that determines its lifetime. There are 2 computing space durations: static, and automatic. A network declared with *static* computing space duration is created only once and exists till termination of the entire program. A new instance of a network declared with *automatic* computing space duration is created on each entry into the block in which it is declared. The network is discarded when execution of the block ends.

Now, let us consider a simple mpC program computing the dot product of two vectors. The program is correct but not good in the sense of efficiency.

```
/*1*/  nettype Star(n) {
/*2*/    coord I=n;
/*3*/    node { default: scalar;};
/*4*/    link { I>0: [0]<->[i];};
/*5*/    parent [0];
/*6*/  };
/*7*/  #define N 100
/*8*/  void [*]main()
/*9*/  {
/*10*/    double [host]x[N];
/*11*/    double [host]y[N];
/*12*/    double [host]z;
/*13*/    double sqrt();
/*14*/    .../*Input of x and y */
/*15*/    {
/*16*/      net Star(N) s;
/*17*/      double [s]dx, sudd, sods;
/*18*/      desex[];
/*19*/      day[];
/*20*/      dz=dx*dy;
/*21*/      z=[host]dz[+];
/*22*/      z=([host]sqrt)(z);
/*23*/    }
/*24*/    .../* Output of z */
/*25*/  }
```

The program includes 2 functions - `main` defined here and library function `sqrt`. Lines 8-25 contain a definition of `main`. Lines 10-12 contain definitions of arrays `x`, `y` and variable `z` all belonging to the virtual host-processor. Line 13 contains a declaration of function identifier `sqrt`.

In general, mpC allows 3 kinds of functions. Here, functions of two kinds are used: `main` is a *basic* function, and `sqrt` is a *nodal* function.

A call to *basic function* is executed on the entire computing space. Its arguments should either belong to the virtual host-processor or be distributed over the entire computing space, and its value should be distributed over the entire computing space. In contrast to other kinds of function, a basic function can define networks. In line 8, construct `[*]`, placed just before the function identifier, specifies that `main` is an identifier of basic function.

Nodal function can be executed completely by any one virtual processor. Only local data objects of the executing virtual processor may be defined in such a function. In addition, the corresponding component of an externally-defined distributed data object can be used in the function. A declaration of nodal function (e.g., in line 13) does not need any additional specifiers.

Line 16 defines the automatic network `s` with the virtual host-processor as a parent.

Line 17 defines 3 automatic variables `dx`, `dy`, and `dz` all distributed over `s`.

Line 18 contains unusual unary postfix operator `[]`. In general, its operand should either designate an array or be a pointer. In this case, expression `x[]` designates array `x` as a whole, and the statement in line 18 scatters elements of array `x` to components of distributed variable `dx`.

Similarly, the statement in line 19 scatters elements of array `y` to components of distributed variable `dy`.

The statement in line 20 is also executed on network `s`. But unlike 2 previous statements, its execution does not need any communications between virtual processors constituting network `s`.

In fact, this statement is divided into a set of independent undistributed statements each of which is executed by the corresponding virtual processor using the corresponding data components. Such statement are called *asynchronous* statements. In particular, this statement multiplies (in parallel) components of `dx` and `dy` and assigns the result to components of `dz`.

In line 21, the result of postfix unary operator `[+]` is distributed over `s`. All its components are equal to the sum of all components of operand `dz`. Here, the result of prefix unary operator `[host]` is the component of its operand belonging to the virtual host-processor. So, the statement in line 21 assigns the sum of all components of `dz` to `z`.

Finally, line 22 calls to nodal function `sqrt` on the virtual host-processor and assigns the value returned to `z`.

To support modular parallel programming as well as the writing of libraries of parallel programs, so-called network functions are introduced in addition to basic and nodal functions.

3. The mpC programming environment

Currently, the mpC programming environment includes a compiler, a run-time support system (RTSS), a library, and a command-line user interface.

The compiler translates a source mpC program into the ANSI C program with calls to functions of RTSS.

RTSS manages the computing space which consists of a number of processes running over target DMM as well as provides communications. It has a precisely specified interface and encapsulates a particular communication package (currently, a small subset of MPI). It ensures platform-independence of the rest of system components.

The library consists of a number of functions that support debugging mpC programs as well as provide some low-level efficient facilities.

The command-line user interface consists of a number of shell commands supporting the creation of a virtual parallel machine and the execution of mpC programs on the machine. While creating the machine, its topology is detected by a topology detector running a special benchmark and saved in a file used by RTSS.

Our compiler uses optionally either the SPMD model of target code, when all processes constituting a target message-passing program run identical code, or a quasi-SPMD model, when it translates a source mpC file into 2 separate target files - the first for the virtual host-processor and the second for the rest of virtual processors.

All processes constituting the target program are divided into 2 groups - the special process called *dispatcher* playing the role of the computing space manager, and general processes called *nodes* playing the role of virtual processors of the computing space. The special node called *host* is separated. The dispatcher works as a server accepting requests from nodes. The dispatcher does not belong to the computing space.

In the target program, every network or subnetwork of the source mpC program is represented by a set of nodes called *region*. At any time of the target program running, any node is either free or hired in one or several regions. Hiring nodes in created regions and dismissing them are responsibility of the dispatcher. The only exception is the pre-hired host-node representing the mpC pre-defined virtual host-processor. Thus, just after initialization, the computing space is represented by the host and a set of temporarily free (unemployed) nodes.

Creation of the network region involves the parent node, the dispatcher and all free nodes. The parent node sends a creation request containing the necessary information about the network

topology to the dispatcher. Based on this information and the information about the topology of the virtual parallel machine, the dispatcher selects the most appropriate set of free nodes. After that, it sends to every free node a message saying whether the node is hired in the created region or not. Deallocation of network region involves all its members as well as the dispatcher.

The dispatcher keeps a queue of creation requests that cannot be satisfied immediately but can be served in the future. It implements some strategy of serving the requests aimed at minimization of the probability of occurring a deadlock. The dispatcher detects such a situation when the sum of the number of free nodes and the number of such hired nodes that could be released is less than the minimum number of free nodes required by a request in the queue. In this case, it terminates the program abnormally specifying a deadlock.

4. Irregular applications

4.1. Programming in mpC

Let us consider an irregular application simulating the evolution of a system of stars in a galaxy (or set of galaxies) under the in^{a}uence of Newtonian gravitational attraction.

Let our system consist of a number of large groups of bodies. It is known, that since the magnitude of interaction between bodies falls off rapidly with distance, the effect of a large group of bodies may be approximated by a single equivalent body, if the group of bodies is far enough away from the point at which the effect is being evaluated. Let it be true in our case. So, we can parallelize the problem, and our application will use a few virtual processors, each of which updates data characterizing a single group of bodies. Each virtual processor holds attributes of all the bodies constituting the corresponding group as well as masses and centers of gravity of other groups. The attributes characterizing a body include its position, velocity and mass.

Finally, let our application allow both the number of groups and the number of bodies in each group to be defined in run time.

The application implements the following scheme:

```
Initializing the galaxy
        on the virtual host-processor
Creation of the network
Scattering groups over
        virtual processors
Parallel computing masses of groups
Interchanging the masses among
        virtual processors
while(1) {
  Visualization of the galaxy
        on the virtual host-processor
  Parallel computation of centers of
        gravity of groups
  Interchanging the centers among
        virtual processors
  Parallel updating groups
  Gathering groups
        on the virtual host-processor
}
```

The corresponding mpC program looks as follows:

```
#define DELTA 3600.0
#define INTERVAL 3

/*The maximum number of groups*/
#define MaxGs 30

/*The maximum number of bodies in a group*/
#define MaxBs 600

typedef double Triplet[3];
typedef
  struct {Triplet pos; Triplet v; double m;}
  Body;

/*The number of groups*/
int [host]M;

/*The numbers of bodies in groups*/
int [host]N[MaxGs];

repl dM, dN[MaxGs];

/*The galaxy timer*/
double [host]t;

/*Bodies of a galaxy*/
Body (*[host]Galaxy[MaxGs])[MaxBs];


nettype GalaxyNet(m, n[m]) {
  coord I=m;
  node { I>=0: fast*n[I] scalar;};
  link (J=m){
    J>0: length*(-1) [J]->[0];
    J>0: length*1    [I]->[J];
  };
};


void [host]Input(), UpdateGroup();

void [host]VisualizeGalaxy();

void [*]Nbody(char *[host]infile)
{
  /*Initializing Galaxy, M and N*/
  Input(infile);

  /*Broadcasting the number of groups*/
  dM=M;

  /*Broadcasting the numbers of bodies*/
  /*in groups*/
  dN[]=N[];
```

```
  {
    net GalaxyNet(dM,dN) g;
    int [g]myN, [g]mycoord;
    Body [g]Group[MaxBs];
    Triplet [g]Centers[MaxGs];
    double [g]Masses[MaxGs];
    repl [g]i;
    void [net GalaxyNet(m, n[m])]Mintegrity
      (double (*)[MaxGs]);
    void [net GalaxyNet(m, n[m])]Cintegrity
      (Triplet (*)[MaxGs]);

   mycoord = I coordof body_count;
   myN = dN[mycoord];

   /*Scattering groups*/
   for(i=0; i<[g]dM; i++)
     [g:I==i]Group[] = (*Galaxy[i])[];

   for(i=0; i<myN; i++)
     Masses[mycoord]+=Group[i].m;
   ([([g]dM,[g]dN)g])Mintegrity(Masses);
   while(1) {
     if(((int)(t/DELTA))%INTERVAL==0)
       VisualizeGalaxy();
     Centers[mycoord][]=0.0;
     for(i=0; i<myN; i++)
       Centers[mycoord][] +=
         (Group[i].m/Masses[mycoord])*
         (Group[i].pos)[];
     ([([g]dM,[g]dN)g])Cintegrity(Centers);
     ([g]UpdateGroup)(Centers, Masses,
                      Group, [g]dM);
     t+=DELTA;
     if(((int)(t/DELTA))%INTERVAL==0)
       /*Gathering groups*/
       for(i=0; i<[g]dM; i++)
         (*Galaxy[i])[]=[g:I==i]Group[];
   }
 }
}

void [net GalaxyNet(m,n[m]) p] Mintegrity
  (double (*Masses)[MaxGs])
{
  double MassOfMyGroup;
  repl i, j;
  MassOfMyGroup=(*Masses)[I coordof i];
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      [p:I==i](*Masses)[j] =
        [p:I==j]MassOfMyGroup;
}
```

```
void [net GalaxyNet(m,n[m]) p] Cintegrity
  (Triplet (*Centers)[MaxGs])
{
  Triplet MyCenter;
  repl i, j;
  MyCenter[] = (*Centers)[I coordof i][];
  for(i=0; i<m; i++)
    for(j=0; j<m; j++)
      [p:I==i](*Centers)[j][] =
        [p:I==j]MyCenter[];
}
```

This mpC source file contains the following external definitions:

- definitions of variables `M`, `t` and arrays `N`, `Galaxy` all belonging to the virtual host-processor;
- a definition of variable `dM` and array `dN` both replicated over the entire computing space;
- a definition of network type `GalaxyNet`;
- a definition of basic function `Nbody` with one formal parameter `infile` belonging to the virtual host-processor;
- definitions of network functions `Mintegrity` and `Cintegrity`.

In general, a *network function* is called and executed on some network or subnetwork, and its value is also distributed over this region of the computing space. The header of the definition of the network function either specifies an identifier of a global static network or subnetwork, or declares an identifier of the network being a special formal parameter of the function. In the first case, the function can be called only on the specified region of the computing space. In the second case, it can be called on any network or subnetwork of a suitable type. In any case, only the network specified in the header of the function definition may be used in the function body. No network can be declared in the body. Only data objects belonging to the network specified in the header may be defined in the body. In addition, corresponding components of an externally-defined distributed data object may be used. Unlike basic functions, network functions (as well as nodal functions) can be called in parallel.

Network functions `Input` and `VisualizeGalaxy`, both associated with the virtual host-processor, as well as the nodal function `UpdateGroup` are declared and called here.

Automatic network `g`, executing most of computations and communications, is defined in such a way, that it consists of `M` virtual processors, and the relative performance of each processor is characterized by the number of bodies in the group which it computes.

So, the more powerful is the virtual processor, the larger group of bodies it computes, and the more intensive is the data transfer between two virtual processors, the shorter link connects them (length specifier `length*(-1)` specifies a shorter link than `length*1` does).

The mpC programming environment bases on this information to map the virtual processors constituting network `g` into the processes constituting the entire computing space in the most appropriate way. Since it does it in run time, the user does not need to recompile this mpC program, to port it to another DMM.

The result of the binary operator `coordof` (in the first statement of the inner block of function `Nbody`) is an integer value distributed over `g`, each component of which is equal to the value of coordinate `I` of the virtual processor to which the component belongs. The right operand of operator `coordof` is not evaluated and used only to specify a region of the computing space. Note, that coordinate variable `I` is treated as an integer variable distributed over the region.

Call expression `([g]UpdateGroup)(....)` causes parallel execution of nodal function

`UpdateGroup` on each of virtual processors of network `g`. It is meant, that function name `UpdateGroup` is converted to a pointer-to-function distributed over the entire computing space, and operator `[g]` cuts from this pointer a pointer distributed over `g`. So, the value of expression `[g]UpdateGroup` is a pointer-to-function distributed over `g`. Therefore, expression `([g]UpdateGroup)(....)` denotes a distributed call to a set of undistributed functions.

Network functions `Mintegrity` and `Cintegrity` have 3 special formal parameters. Network parameter `p` denotes the network executing the function. Parameter `m` is treated as a replicated over `p` integer variable, and parameter `n` is treated as a pointer to the initial member of an integer unmodifiable `m`-member array replicated over `p`. The syntactic construct `([(dM,dN)g])`, placed on the left of the name of the function called in the call expressions in function `Nbody`, just specifies the actual arguments corresponding to the special formal parameters.

4.2 Experimental results

We compared the running time of our mpC program to its carefully written MPI counterpart. We use 3 workstations - SPARCstation 5 (hostname `gamma`), SPARCclassic (`omega`), and SPARCstation 20 (`alpha`), connected via 10Mbits Ethernet. There were 23 other computers in the same segment of the local network. We used LAM MPI version 5.2 [12] as a particular communication platform.

The computing space of the mpC programming environment consists of 15 processes, 5 processes running on each workstation. The dispatcher runs on `gamma` and uses the following relative performances of the workstations obtained automatically upon the creation of the virtual parallel machine: 1150 (`gamma`), 331 (`omega`), 1662 (`alpha`).

The MPI program is written in such a way to minimize communication overheads. All our experiments deal with 9 groups of bodies. We map 3 MPI processes to `gamma`, 1 process to `omega`, and 5 processes to `alpha`, providing the optimal mapping if the numbers of bodies in these groups are equal to each other.

The first experiment compares the mpC and MPI programs for homogeneous input data when all groups consist of the same number of bodies. Figure1 shows the running time of both programs simulating 15 hours of the galaxy evolution depending on the number of bodies in groups.

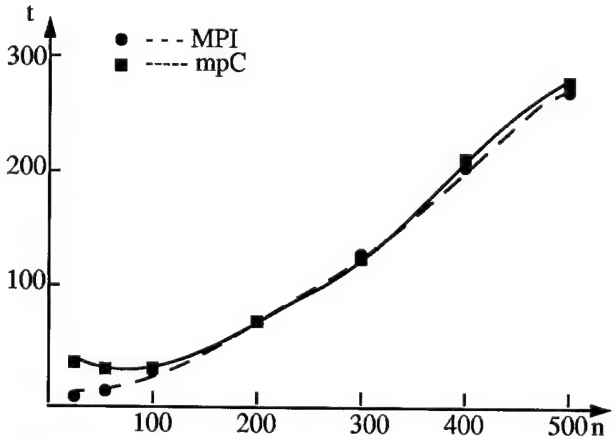

Figure 1. Running time of the MPI and mpC programs for homogenous input data.

In fact, it shows how much we pay for the usage of mpC instead of pure MPI. One can see that

the running time of the MPI program consists about 95-97% of the running time of the mpC program. That is, in this case we loose 3-5% of performance.

The second experiment compares these programs for heterogeneous input data. Let our groups consist of 10, 10, 10, 100, 100, 100, 600, 600, and 600 bodies correspondingly.

The running time of the mpC program does not depend on the order of the numbers. In any case, the dispatcher selects:

- 4 processes on `gamma` for virtual processors of network `g` computing two 10-body groups, one 100-body group, and one 600-body group;
- 3 processes on `omega` for virtual processors computing one 10-body group and two 100-body groups;
- 2 processes on `alpha` for virtual processors computing two 600-body groups.

The mpC program takes 94 seconds to simulate 15 hours of the galaxy evolution.

The running time of the MPI program essentially depends on the order of these numbers. It takes from 88 to 391 seconds to simulate 15 hours of the galaxy evolution depending on the particular order. Figure 2 shows the relative running time of the MPI and mpC programs for different permutations of these numbers. All possible permutations can be broken down into 24 disjoint subsets of the same power in such a way that if two permutations belong to the same subset, the corresponding running time is equal to each other. Let these subsets be numerated so that the greater number the subset has, the longer time the MPI program takes. In figure 2, each such a subset is represented by a bar, the height of which is equal to the corresponding value of t_{MPI}/t_{mpC}.

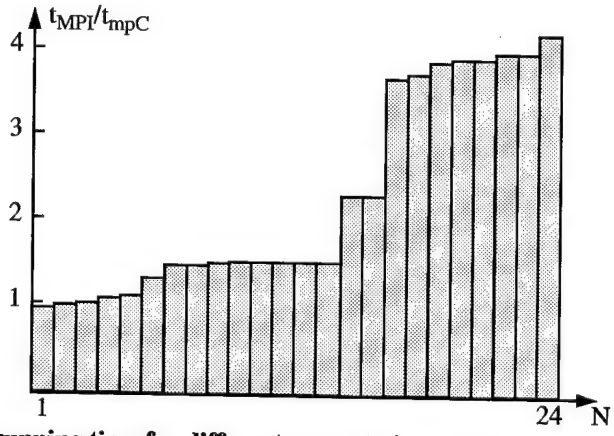

Figure 2. The relative running time for different permutations of the numbers of bodies in groups.

One can see that almost for all input data the running time of the MPI program exceeds (and often, essentially) the running time of the mpC program.

5. Regular applications

5.1 Programming in mpC

Let us consider a regular application multiplying 2 dense square `nxn` matrices `X` and `Y`.

Our mpC program will use a number of virtual processors, each of which computes a number of rows of the resulting matrix `Z`. Both dimension `n` of matrices and the number of virtual processors involved in computations are defined in run time. So, our application implements the following

scheme:

```
Initializing X and Y
    on the virtual host-processor
Creating a network
Scattering rows of X over
    virtual processors of the network
Broadcasting Y over
    virtual processors of the network
Parallel computing submatrices of Z
Gathering the resulting matrix Z
    on the virtual host-processor
```

The corresponding mpC program looks as follows:

```
/*1*/   nettype SimpleNet(n) {
/*2*/    coord I=n;
/*3*/   };

/*4*/   nettype Star(m, n[m]) {
/*5*/    coord I=m;
/*6*/    node {I>=0: fast*n[I] scalar;};
/*7*/    link {I>0:  [I]->[0], [0]->[I];};
/*8*/    parent [0];
/*9*/   };

/*10*/  void [*]MxM(float *x, float *y,
/*11*/             float *z, repl n) {
/*12*/   repl double *powers;
/*13*/   repl nprocs, nrows[MAXNPROCS], n;
/*14*/
/*15*/   MPC_Processors_static_info
/*16*/                 (&nprocs,&powers);
/*17*/   Partition(nprocs,powers,nrows,n);
/*18*/   {
/*19*/    net Star(nprocs, nrows) w;
/*20*/    ([([w]nprocs)w])ParMult(
/*21*/     [w]x,[w]y,[w]z,[w]nrows,[w]n);
/*22*/   }
/*23*/  }

/*24*/  void [net SimpleNet(p)v] ParMult(
/*25*/   float *dx, float *dy, float *dz,
/*26*/   repl *r, repl n)
/*27*/  {
/*28*/   repl s=0;
/*29*/   int myn, i;
/*30*/   int *d, *l, c;
/*31*/
/*32*/   myn=r[I coordof r];
/*33*/   ([(p)v])MPC_Bcast(&s, dy, 1,
/*34*/                     n*n, dy, 1);
/*35*/   d=calloc(p, sizeof(int));
/*36*/   l=calloc(p, sizeof(int));
```

```
/*37*/   for(i=0, d[0]=0; i<p; i++) {
/*38*/     l[i]=r[i]*n;
/*39*/    if(i+1<p) d[i+1]=l[i]+d[i];
/*40*/   }
/*41*/   c=l[I coordof c];
/*42*/   ([(p)v])MPC_Scatter(&s, dx ,d,
/*43*/                       l, c, dx);
/*44*/   SeqMult(dx, dy, dz, myn, n);
/*45*/  ([(p)v])MPC_Gather(&s,dz,d,l,c,dz);
/*46*/  }

/*47*/ void SeqMult(float *a, float *b,
/*48*/              float *c, int m, int n)
/*49*/ {
/*50*/  int i, j, k, ixn;
/*51*/  double s;
/*52*/
/*53*/  for(i=0; i<m; i++)
/*54*/   for(j=0, ixn=i*n; j<n; j++) {
/*55*/    for(k=0, s=0.0; k<n; k++)
/*56*/     s+=a[ixn+k]*(double)(b[k*n+j]);
/*57*/    c[ixn+j]=s;
/*58*/   }
/*59*/ }


/*60*/  void Partition(int p, double *v,
/*61*/                 int *r, int n)
/*62*/  {
/*63*/   int sr, i;
/*64*/   double sv;
/*65*/
/*66*/   for(i=0, sv=0.0; i<p; i++)
/*67*/    sv+=v[i];
/*68*/   for(i=0, sr=0; i<p; i++) {
/*69*/    r[i]=(int)(v[i]/sv*n);
/*70*/    sr+=r[i];
/*71*/   }
/*72*/   if(sr!=n) r[0]+=n-sr;
/*73*/  }
```

Formal parameters x, y, and z of basic function MxM are distributed over the entire computing space, and parameter n is replicated over the entire computing space. It is meant that n holds the dimension of matrices. It is also meant that x points to nxn-member array, and the component of this distributed array belonging to the virtual host-processor holds matrix X. Similarly, [host]y points to an array holding matrix Y, and [host]z points to an array holding resulting matrix Z.

Lines 15-16 calls to library nodal function MPC_Processors_static_info on the entire computing space returning the number of actual processors and their relative performances. So, after this call replicated variable nprocs will hold the number of actual processors, and replicated array powers will hold their relative performances.

Line 17 calls to nodal function Partition on the entire computing space. Based on relative performances of actual processors, this function computes how many rows of the resulting matrix

will be computed by every actual processor. So, after this call `nrows[i]` will hold the number of rows computed by `i`-th actual processor.

Line 19 defines automatic network `w`. Its type is defined completely only in run time. Network `w`, which executes the rest of computations and communications, is defined in such a way, that the more powerful the virtual processor, the greater number of rows it computes. The mpC environment will ensure the optimal mapping of the virtual processors constituting `w` into a set of processes constituting the entire computing space. So, just one process from processes running on each of actual processors will be involved in multiplication of matrices, and the more powerful the actual processor, the greater number of rows its process will compute.

Lines 20-21 call to network function `ParMult` on network `w`. In this call, topological argument `[w]nprocs` specifies a network type as an instance of parametrized network type `SimpleNet`, and network argument `w` specifies a region of the computing space treated by function `ParMult` as a network of this type.

In lines 24-26, the header of the definition of function `ParMult` declares identifier `v` of a network being a special network formal parameter of the function. Since network `v` has a parametrized type, topological parameter `p` is also declared in this header. In the function body, special formal parameter `p` is treated as an unmodifiable variable of type `int` replicated over network formal parameter `v`. The rest of formal parameters (regular formal parameters) of the function are also distributed over `v`.

Actually, `p` holds the number of virtual processors in network `v`, `n` holds the dimension of matrices, `r` points to `p`-member array, `i`-th element of which holds the number of rows of the resulting matrix that `i`-th virtual processor of network `v` computes. Each component of `dy` points to an array to contain `nxn` matrix `Y`. Each component of `dz` points to an array to contain the rows of `Z` computed on the corresponding virtual processor of `v`. Each component of `dx` points to an array to contain the rows of `X` used in computations on the corresponding virtual processor. In addition, throughout the function execution the components of `dx`, `dy`, `dz` belonging to the parent of network `v` are reputed to point to arrays holding matrices `X`, `Y` and `Z` correspondingly.

Line 28 defines variable `s` replicated over `v`. Lines 29-30 define variables `myn`, `i`, `d`, `l` and `c` all distributed over `v`.

After execution of the asynchronous statement in line 32, each component of `myn` will contain the number of rows of the resulting matrix that computes the corresponding virtual processor.

Lines 33-34 call to so-called *embedded* network function `MPC_Bcast` which is declared in a standard mpC header as follows:

```
int [net SimpleNet(n)] MPC_Bcast(
    repl const *coordinates_of_source,
    void *source_buffer,
    const source_step,
    repl const count,
    void *destination_buffer,
    const destination_step);
```

This call broadcasts matrix `Y` from the parent of `v` to all virtual processors of `v`. As a result, each component of the distributed array pointed by `dy` will contain this matrix.

An embedded network function looks like a library network function, but a compiler knows its semantics. In particular, it will generate different code for different types of arguments corresponding to source and destination buffers.

Statements in lines 35-40 are asynchronous. They form two `p`-member arrays `d` and `l` distributed over `v`. After their execution, `l[i]` will hold the number of elements in the portion of the

resulting matrix which is computed by the i-th virtual processor of v, and d[i] will hold the displacement which corresponds to this portion in the resulting matrix. Equivalently, l[i] will hold the number of elements in the portion of matrix X which is used by i-th virtual processor of v, and d[i] will hold the displacement which corresponds to this portion in matrix X.

The statement in line 41 is also asynchronous. After its execution, each component of c will hold the number of elements in the portion of the resulting matrix which is computed by the corresponding virtual processor (equivalently, the number of elements in the portion of matrix X which is used by this virtual processor).

Lines 42-43 call to embedded network function MPC_Scatter which is declared as follows:

```
int [net SimpleNet(n) w] MPC_Scatter(
    repl const *coordinates_of_source,
    void *source_buffer,
    const *displacements,
    const *sendcounts,
    const receivecount,
    void *destination_buffer);
```

This call scatters matrix X from the parent of v to all virtual processors of v. As a result, each component of dx will point to an array containing the corresponding portion of matrix X.

Line 44 calls to nodal function SeqMult on v, computing the corresponding portions of the resulting matrix on each of its virtual processors in parallel (SeqMult implements traditional sequential algorithm of matrix multiplication).

Finally, line 45 calls to embedded network function MPC_Gather which is declared as follows:

```
int [net SimpleNet(n) w] MPC_Gather(
  repl const *coordinates_of_destination,
  void *destination_buffer,
  const *displacements,
  const *receivecounts,
  const sendcount,
  void *source_buffer);
```

This call gathers resulting matrix Z each virtual processor of v sending its portion of the result to the parent of v.

5.2 Experimental results

We measured the running time of our mpC program multiplying two dense square matrices. We used three Sun SPARCstations 5 (hostnames gamma, beta, and delta), SPARCclassic (omega), and HP 9000-712 (zeta) connected via 10Mbits Ethernet. There were more than 20 other computers in the same segment of the local network.

We used LAM MPI Version 6.0 as a particular communication platform as well as a new improved benchmark for detecting relative performances of workstations. In addition, all executables, which took part in the experiment, were generated by GNU C compiler with optimization option -O2.

Eight virtual parallel machines were created:

- g consisting of gamma (its relative performance detected during the creation of this virtual parallel machine was equal to 324);
- gd consisting of gamma (323), and delta (330);
- gbd consisting of gamma (324), beta (331), and delta (331);
- gbdz consisting of gamma (324), beta (327), delta (330), and zeta (510);

- zg consisting of zeta (510), and gamma (323);
- zgb consisting of zeta (509), gamma (321), and beta (325);
- zgbd consisting of zeta (466), gamma (328), beta (327), and delta (329);
- zo consisting of zeta (506), and omega (147).

The computing space of each of these virtual parallel machines was constituted by 5 processes running on each of workstations (that is, for example, the computing space of gbdz was constituted by 20 processes). As a base of the comparison we used the running time of a sequential C program implementing the same algorithm which was used in function SeqMult.

Table 1 gives the time of running the mpC program on four virtual parallel machines (g, gd, gbd, and gbdz) dependent on the dimension of multiplied matrices, and compares it to the time of running the sequential C program on workstation gamma. Machines g, gd, and gbd are homogeneous ones, meantime machine gbdz is heterogeneous.

Figure 3 illustrates how the mpC program allows to speed up the multiplication of two dense square matrices, if the user starts from single workstation gamma and enhances his computing facilities step by step by means of adding workstations delta, beta and zeta.

Table 1: Time to multiply two nxn matrices (sec)

n	g	g	gd	gbd	gbdz
	C	mpC	mpC	mpC	mpC
100	0.32	0.40	0.53	0.61	0.70
200	2.55	2.61	2.00	1.91	2.05
300	9.33	9.66	6.11	5.25	4.96
400	31.2	32.2	17.9	13.9	11.6
500	54.7	55.6	31.0	23.4	19.0
600	125.	125.	68.0	49.0	37.0
700	196.	196.	106.	75.0	58.0
800	320.	323.	172.	123.0	88.

Note, that the running time of the mpC program substantially depends on the network load. We monitored the network activity during our experiments. We have observed up to 32 collisions per second. The collisions occurred more often during broadcasting large data portions. The collisions resulted in visible degradation of the network bandwidth.

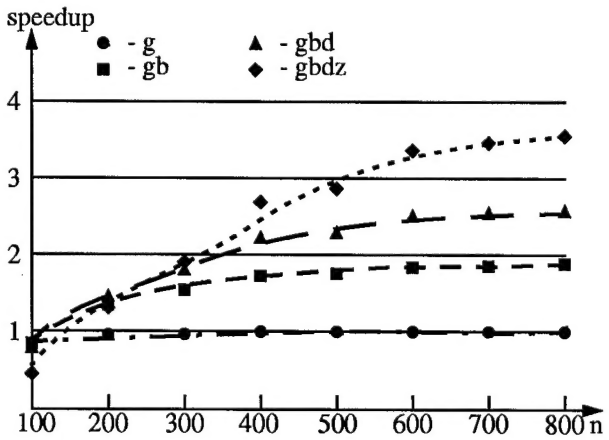

Figure 3. Speedups computed relative to sequential code running on workstation gamma.

Table 2 compares contribution of communications and computations in the total running time of the mpC program (results for `gbdz` are presented). The first column shows matrix dimensions, an the second one shows percentage of communications in the total running time.

Table 2:Contribution of communications in the total running time (`gbdz`)

n	Communications (%)
100	40
200	55
300	48
400	38
500	35
600	26
700	24
800	21

Communications in our mpC program consist of three parts: scattering matrix `X`, broadcasting matrix `Y`, and gathering the resulting matrix. Table 3 compares contribution of each of these parts in the total communication time (for the `gbdz` virtual parallel machine).

While analyzing the presented results, it is necessary to take into account some peculiarities of both the implementation of MPI, which we used, and our local network.

Our local network does not support fast communications. It is based on 10Mbits Ethernet and uses old-fashioned network equipment. In addition, there are 26 computers in our segment of the network connected via cascade of 4 hubs. To characterize our network, it is enough to say that `ftp` transfers data from `gamma` to `alpha` at the rate of 300-400Kbytes/s. It means that real bandwidth of our network is about 25-30% of its maximum bandwidth.

Table 3:Contribution of broadcast, scatter, and gather in the total communication time (`gbdz`)

n	bcast	scatter	gather
100	70%	18%	12%
200	78%	11%	11%
300	78%	10%	12%
400	79%	10%	11%
500	79%	10%	11%
600	79%	10%	11%
700	79%	10%	11%
800	76%	13%	11%

On the other hand, LAM MPI Version 6.0 ensures sending large "oating arrays at the rate of 50-60Kbytes/s. In addition, it doesn't use multicasting facilities of our network when broadcasting.

Nevertheless, even under these conditions, our mpC program has demonstrated good speedup comparing with the sequential C program.

If the implementation of MPI ensured the communication rate comparable with the real bandwidth of the local network and used its multicasting facilities, contribution of communications in the total running time of our mpC program would not exceed 5-7%. If, in addition, we used 100Mbits Ethernet and up-to-date network technologies (for example, replaced hubs with switching devices), contribution of communications in the total running time of the mpC program would not exceed 1-2%. That is, the mpC programming environment can ensure practically ideal speedup of the presented mpC program for up-to-date heterogeneous networks of workstations.

Table 4 gives the time of running the mpC program on four heterogeneous virtual parallel

machines (zg, zgb, zgbd, and zo) dependent on the dimension of multiplied matrices, and compares it to the time of running the sequential C program on workstation zeta.

Table 4: Time to multiply two nxn matrices (sec)

n	z	zg	zgb	zgbd	zo	zo
	C	mpC	mpC	mpC	mpC	MPI
100	0.18	0.43	0.52	0.57	0.67	0.91
200	1.52	1.67	1.70	1.79	2.36	4.29
300	6.80	5.66	5.08	4.90	7.09	14.2
400	17.3	14.2	11.7	11.1	16.4	33.0
500	36.2	26.0	21.0	19.0	32.8	68.0
600	66.8	53.0	41.0	37.0	58.5	120.
700	113.	83.0	64.0	56.0	97.0	200.
800	180.	134.	102.	88.0	152.	306.

In addition, the table compares the mpC program and its manually written MPI counterpart on machine zo.

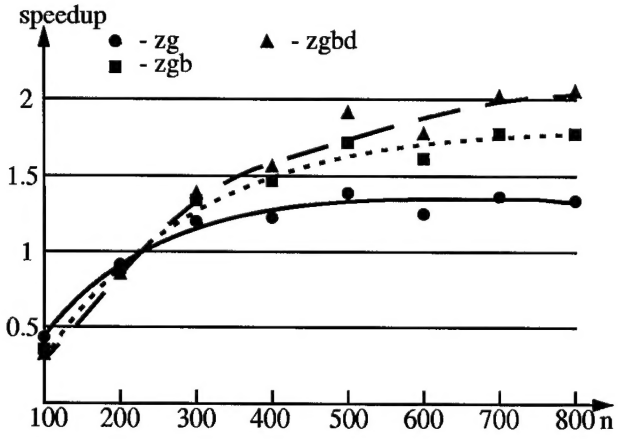

Figure 4. Speedups computed relative to sequential code running on workstation zeta.

Figure 4 illustrates how the mpC program allows to speed up the multiplication of two dense square matrices, if the user starts from single powerful workstation zeta and enhances his computing facilities step by step by means of adding less powerful workstations gamma, beta, and delta. One can see that the mpC programming environment ensures good speedup in this case also.

Another interesting result can be extracted from tables 1 and 4. One can see that the slow network consisting of workstations gamma and delta (virtual parallel machine gd), the performance each of which is about 60% of the performance of workstation zeta, demonstrates a little bit higher performance (when multiplying two dense square matrices) than single workstation zeta.

Finally, figure 5 shows clearly, that even for very heterogeneous distributed memory machine consisting of high-performance HP workstation zeta and low-performance Sun workstation omega, the mpC program allows to utilize its parallel potential, speeding up the multiplication

of two dense square matrices (comparing to the sequential C program running on `zeta`). At the same time, the use of its MPI counterpart, which distributes the workload equally, does not allow to do it slowing down the matrix multiplication essentially.

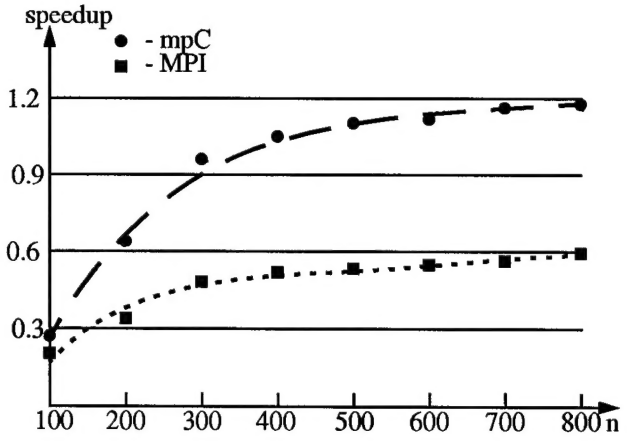

Figure 5. Speedups for MPI and mpC programs both running on machine `zo`.

6. Summary

The key peculiarity of mpC is its advanced facilities for managing such resources of DMMs as processors and links between them. They allow to develop parallel programs for DMMs that once compiled will run efficiently on any particular DMM, because the mpC programming environment ensures optimal distribution of computations and communications over DMM in run time.

The mpC language is a medium-level one. It demands from the user more than high-level parallel languages (say, Fortran D), but much less than MPI or PVM.

Like MPI and PVM, mpC supports efficient programming a particular DMM. Like MPI, the user does not need to rewrite (and, moreover, to recompile) an mpC program to port it to other DMMs.

At the same time, MPI (as well as MPI-2) does not ensure efficient porting to other DMMs, that is, it does not ensure, that a program, running efficiently on a particular DMM, will run efficiently after porting to other DMM. The mpC language and its programming environment do it.

Advantages of mpC are especially clear when programming heterogeneous (irregular) applications or/and programming for heterogeneous DMM.

It makes mpC and its programming environment suitable tools for development of libraries of parallel programs, especially for heterogeneous DMMs.

The paradigm of parallel programming, supported by mpC, foresees explicit specification of a virtual parallel machine executing computations and communications. At the same time, mpC also supports implicit parallel programming, when parallelism is reduced to calls to library basic functions (like function `Nbody` from section 4.1) that just encapsulate parallelism.